A Protected Block Device for Persistent Memory

Feng Chen
Computer Science & Engineering
Louisiana State University

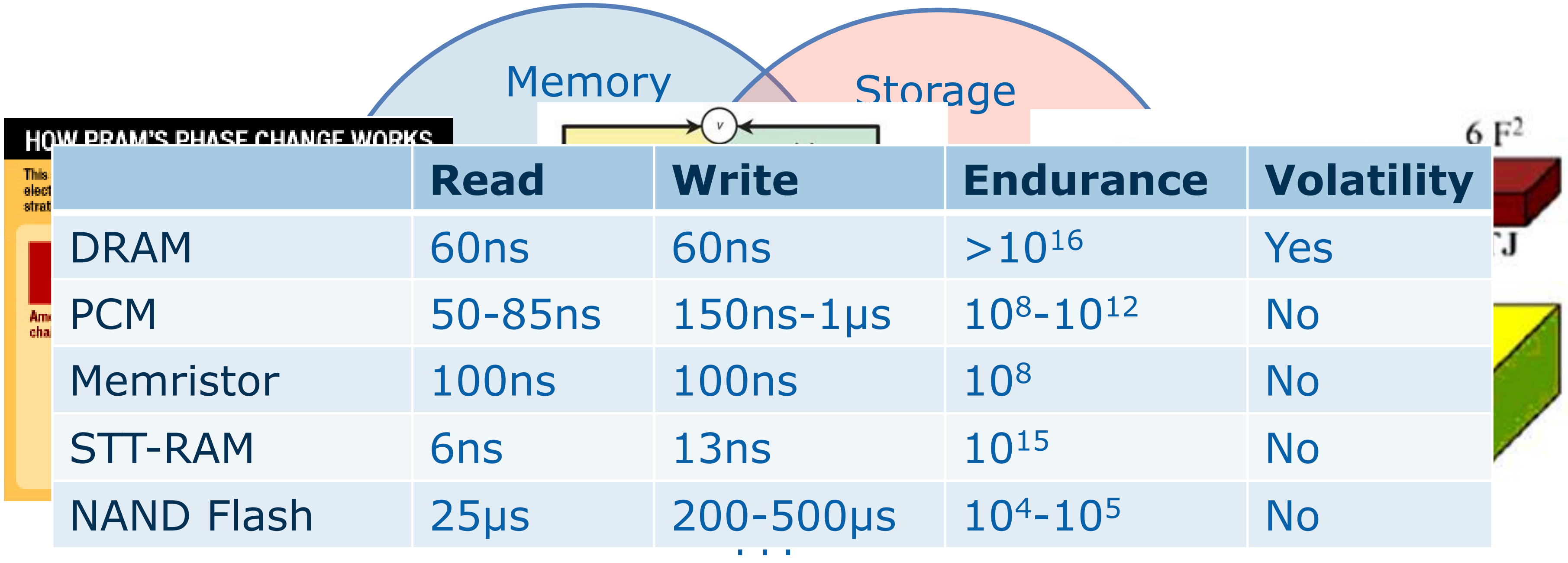
Michael Mesnier
Circuits & Systems Research
Intel Labs

Scott Hahn
Circuits & Systems Research
Intel Labs

Persistent memory (PM)

Unique characteristics

- Memory-like features – fast, byte-addressable
- Storage-like features – non-volatile, relatively endurable



	Read	Write	Endurance	Volatility
DRAM	60ns	60ns	$>10^{16}$	Yes
PCM	50-85ns	150ns-1μs	10^{8}-10^{12}	No
Memristor	100ns	100ns	10^{8}	No
STT-RAM	6ns	13ns	10^{15}	No
NAND Flash	25μs	200-500μs	10^{4}-10^{5}	No

(Protection, persistence)

How should we adopt this new technology in the ecosystem?

Design philosophy

Why not an idealistic approach – redesigning the OS

- Too many implicit assumptions in the existing OS design
- Huge amount of IP asset surrounding the existing eco-system
- Commercial users need to be warmed up to (radical) changes
  - E.g., new programming models (NV-Heaps, CDDS, Mnemosyne)

We need an <u>evolutionary</u> approach to a <u>revolutionary</u> technology

Two basic usage models of PM

Memory based model

- Similar to DRAM (as memory)
- Directly attached to the high-speed memory bus
- PM is managed by memory controller and close to the CPU

Storage based model

- A replacement of NAND flash in SSDs
- Attached to the I/O bus (e.g. SATA, SAS, PCI-E)
- PM is managed by I/O controller and distant from the CPU

Memory model vs. storage model

Compatibility

- Memory model requires changes (e.g., data placement decision)

Performance

- Storage model has lower performance (lower-speed I/O bus)

Protection

- Memory model has greater risk of data corruption (stray pointer writes)

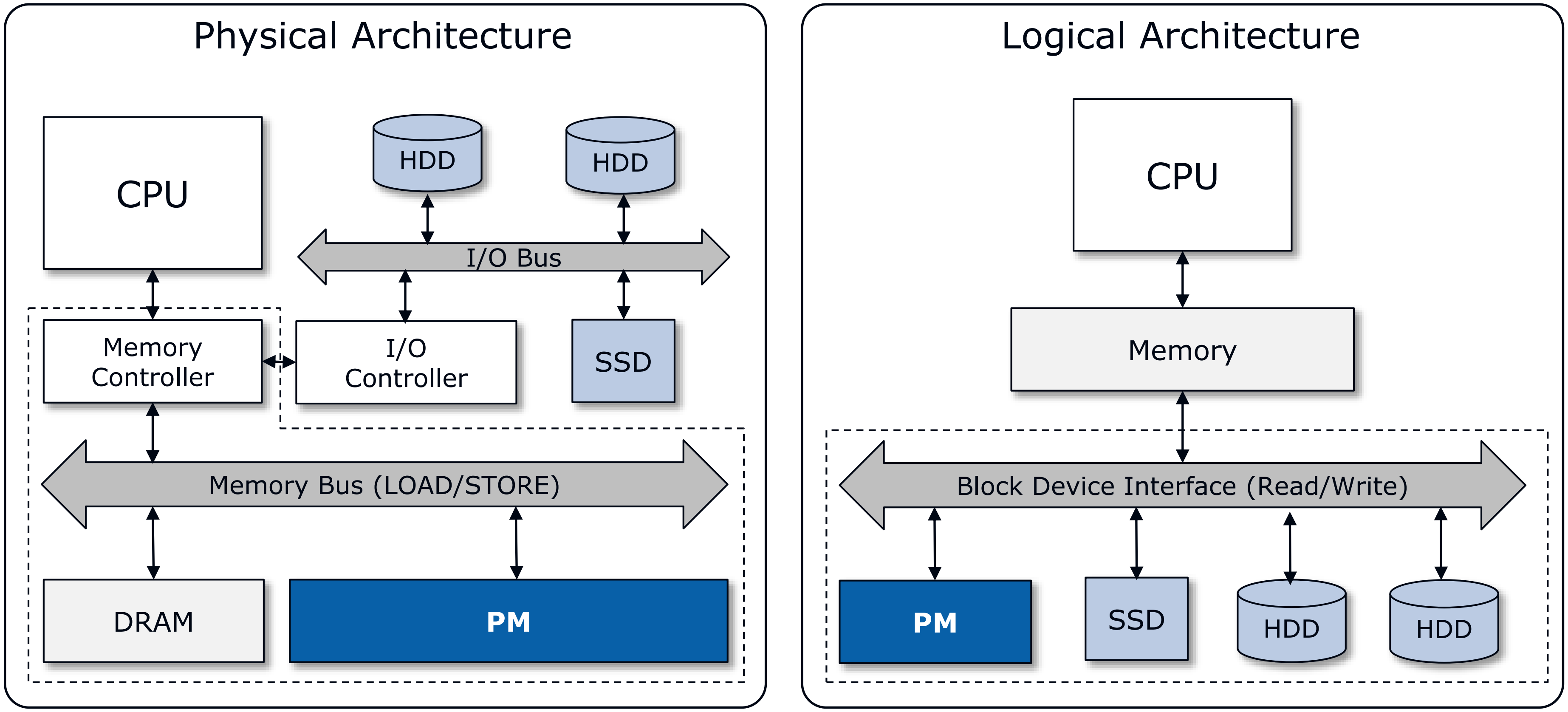
Persistence

- Memory model suffers data loss during power failure (CPU cache effect)

	Performance	Protection	Persistence	Compatibility
Memory model	High	Low	Low	Low
Storage model	Low	High	High	High

How can we get the best of both worlds?

A hybrid memory-storage model for PM



Hybrid PMBD Architecture

Physically ***managed (like memory),*** ***logically*** ***addressed (like storage)***

Benefits of a hybrid PM model

Compatibility

- Block-device interface → no changes to applications or operating systems

Performance

- Physically managed by memory controller → no slow I/O bus involved

Protection

- An I/O model for PM updates → no risk of stray pointer writes

Persistence

- Persistence can be enforced in one entity with persistent writes and barriers

	Performance	**Protection**	**Persistence**	**Compatibility**
Memory model	High	Low	Low	Low
Storage model	Low	High	High	High
Hybrid Model	**High**	**High**	**High**	**High**

System design and prototype

Design goals

Compatibility – minimal OS and no application modification

Protection – protected as a disk drive

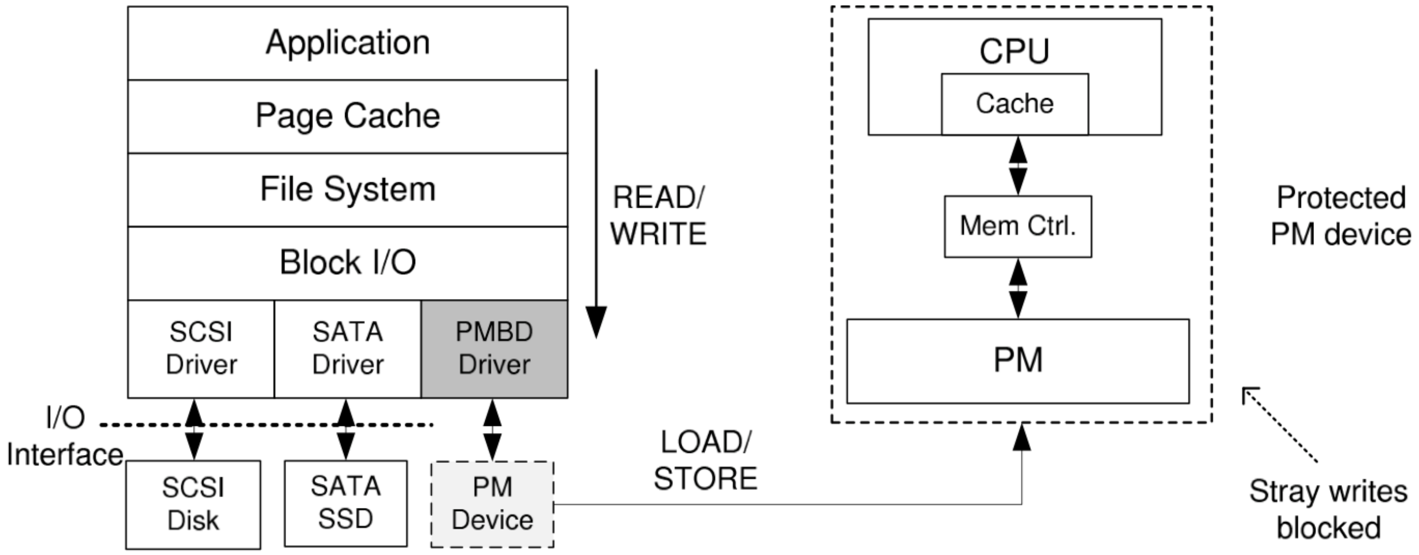
Persistence – as persistent as a disk drive

Performance – close to a memory device

Compatibility via blocks

PM block device (*PMBD*) – No OS, FS, or application modification

- System BIOS exposes a contiguous PM space to the OS
- PMBD Driver provides a generic block device interface (/dev/nva)
- All reads/writes are only allowed through our PM device driver
- Synchronous reads/writes → no interrupts, no context switching



Making PM protected (like disk drives)

Destructively buggy code in kernel

- An example – Intel e1000e driver in Linux Kernel 2.6.27 RC*
- A kernel bug corrupts EEPROM/NVM of Intel Ethernet Adapter

We need to protect the kernel (from itself!)

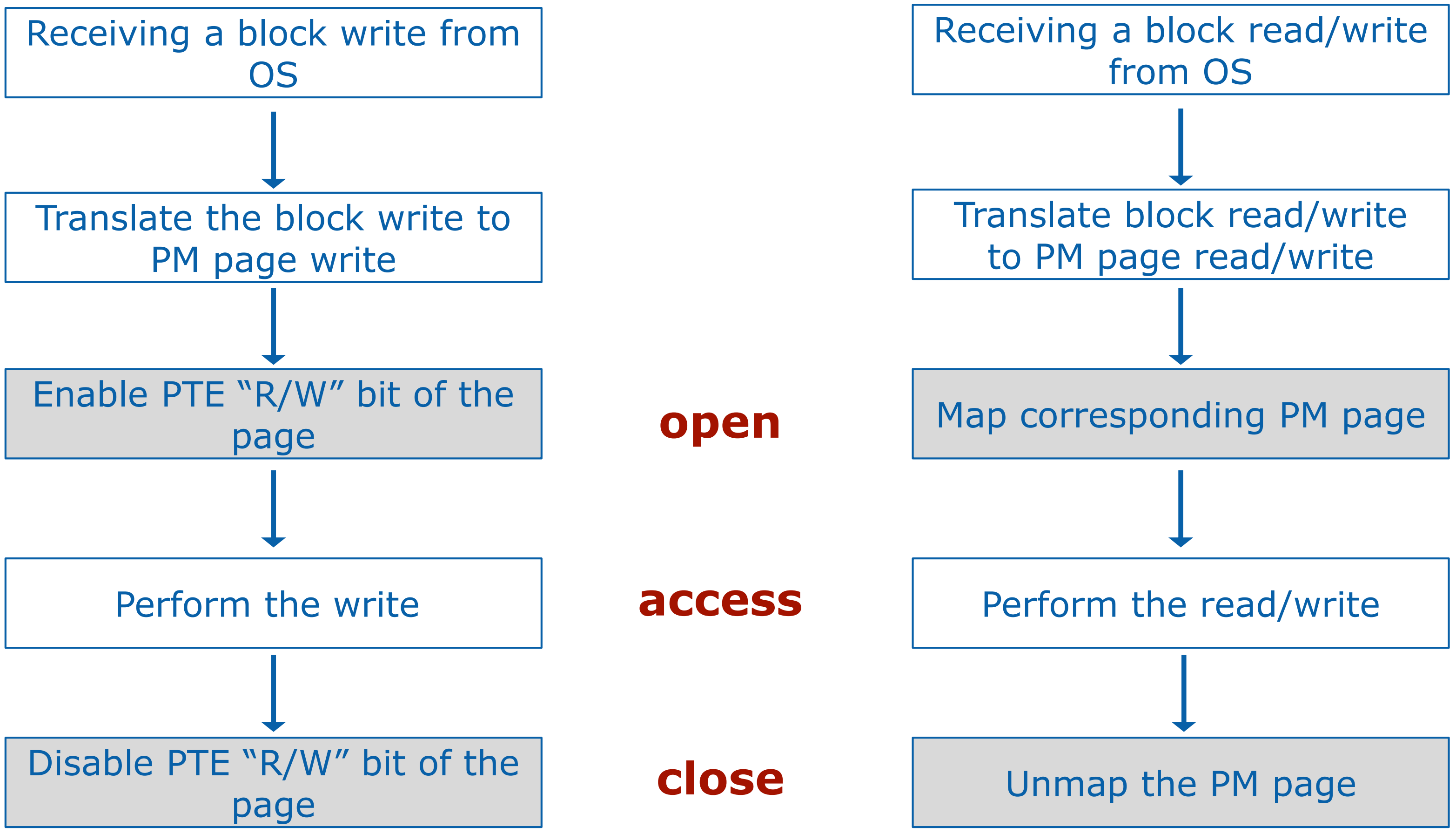
- One address space for the entire kernel
  - All kernel code is inherently trusted (not a safe assumption)
- A stray pointer in the kernel can wipe out all *persistent* data stored in PM
  - No storage "protocol" to block unauthorized memory access

Protection model – Use HW support in existing architecture

- Key rule – PMBD driver is the only entity performing PM I/Os
  - Option 1: Page table based protection (various options explored)
  - Option 2: **Private mapping** based protection (***our recommendation***)

* https://bugzilla.kernel.org/show_bug.cgi?id=11382

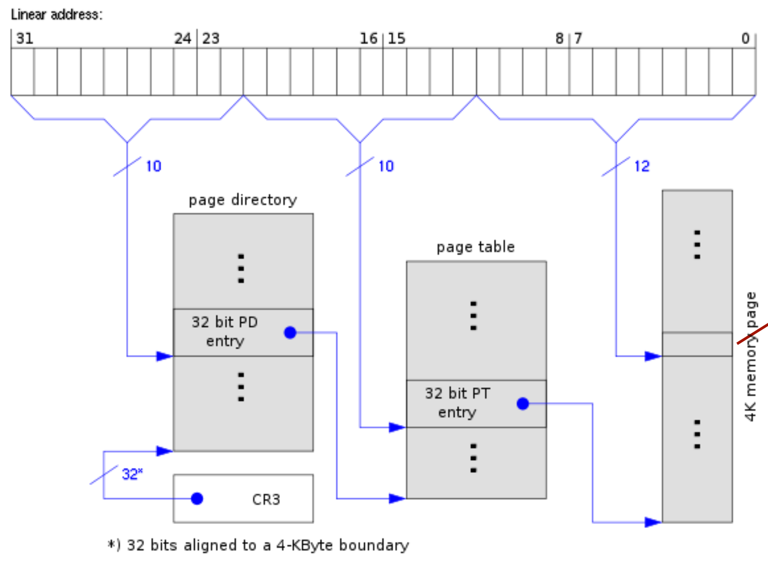
Protection mechanisms


Receiving a block write from OS
Translate the block write to PM page write
Enable PTE "R/W" bit of the page
Perform the write
Disable PTE "R/W" bit of the page
open
access
close
Receiving a block read/write from OS
Translate block read/write to PM page read/write
Map corresponding PM page
Perform the read/write
Unmap the PM page

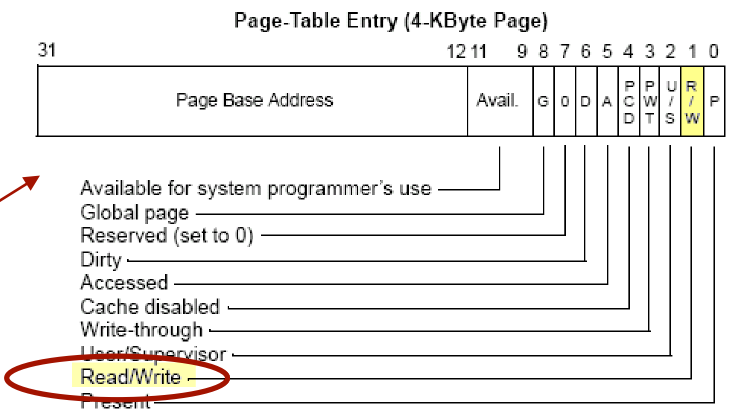
PT-based Protection **Private Mapping Protection**

Protection mechanisms

- **Option 1 – Page table based protection**
  - All PM pages are mapped initially and shared among CPUs
  - Protection is achieved via PTE "R/W" bit control (read-only)
  - **High performance overhead** (TLB shootdowns)



Page Table

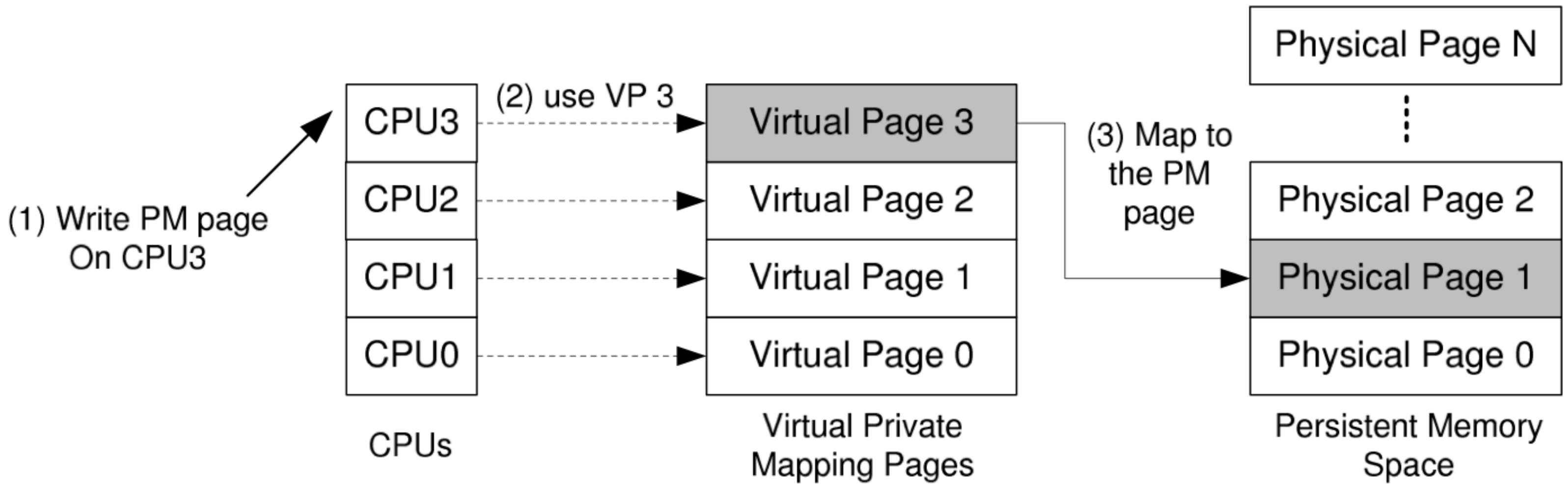


Page Table Entry

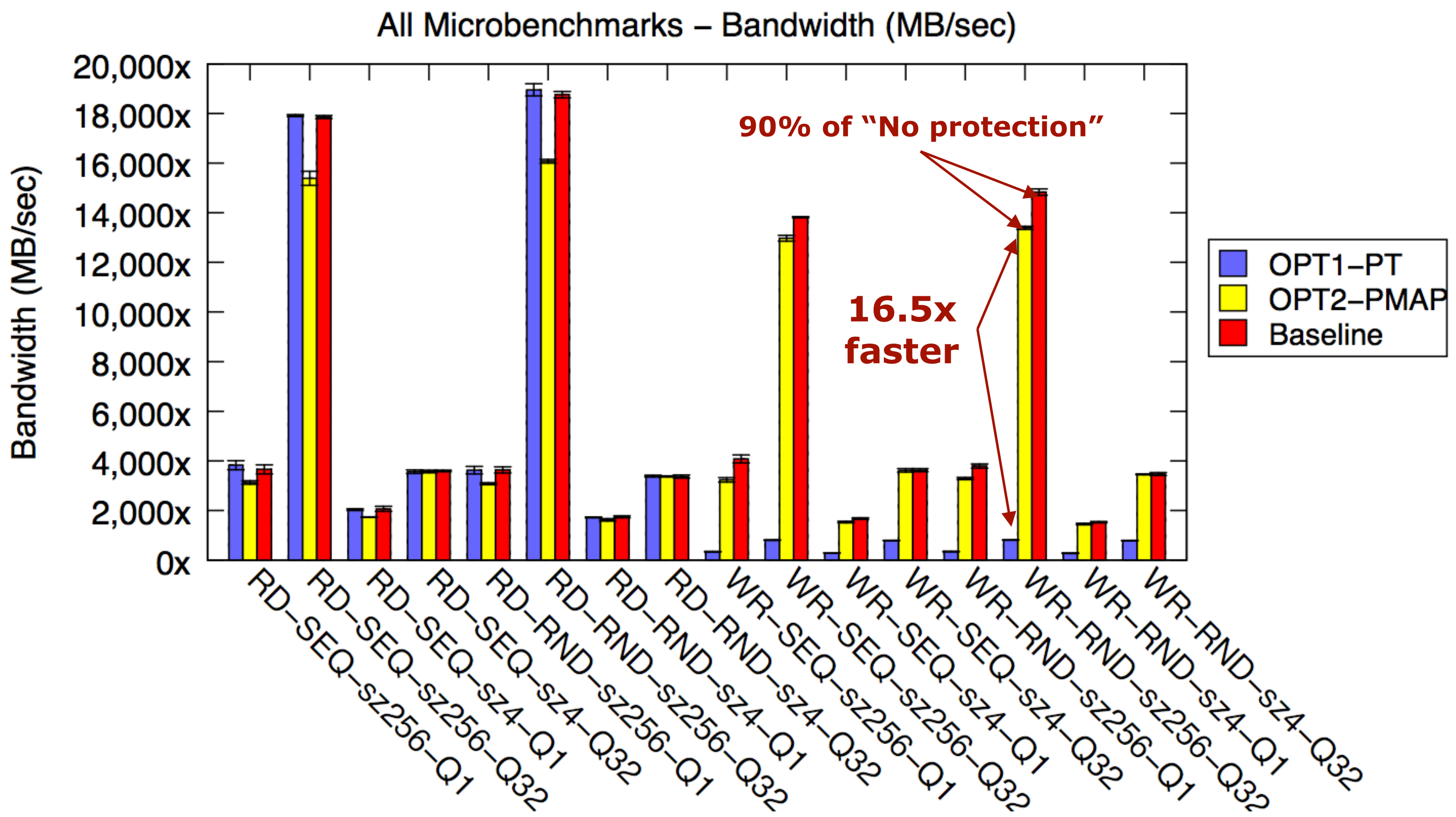
Protection mechanisms

✓ Option 2 – Private (per core) memory mappings

- A PM page is mapped into kernel space only during access
- Multiple mapping entries *p[N]*, each is corresponding to one CPU
- Processes running on CPU *i* use mapping entry *p[i]* to access PM page
- No PTE sharing across CPUs → no TLB shootdown needed



The benefits of private mappings

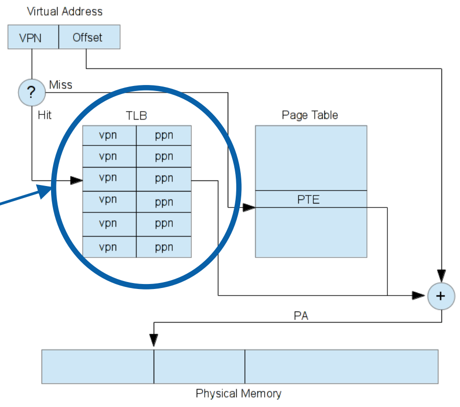


- **Private mapping overhead is small, relative to no protection**
    - Reads (83-100%) and writes (79-99%)
    - Private mapping effectively removes overhead of writes with PT

Other benefits of private mappings

- **Protection for both reads & writes – only authorized I/O**
  - Small window of vulnerability – only active pages visible (one per CPU)
- **scalable O(1) solution – only a page is mapped for each CPU**
  - Small page table size – 1 PTE per core (regardless of PM storage size)
    - e.g., in contrast, 1 TB fully mapped PM requires 2GB for the page table
    - Less memory consumption, shorter driver loading time
  - Small TLB size requirement – only 1 entry is needed for each core
    - Minimized TLB pollution (at most one entry in the TLB)

Small TLB



Private mapping based protection provides high scalability

✓Compatibility
✓Protection
Persistence

Making PM persistent (like disk drives)

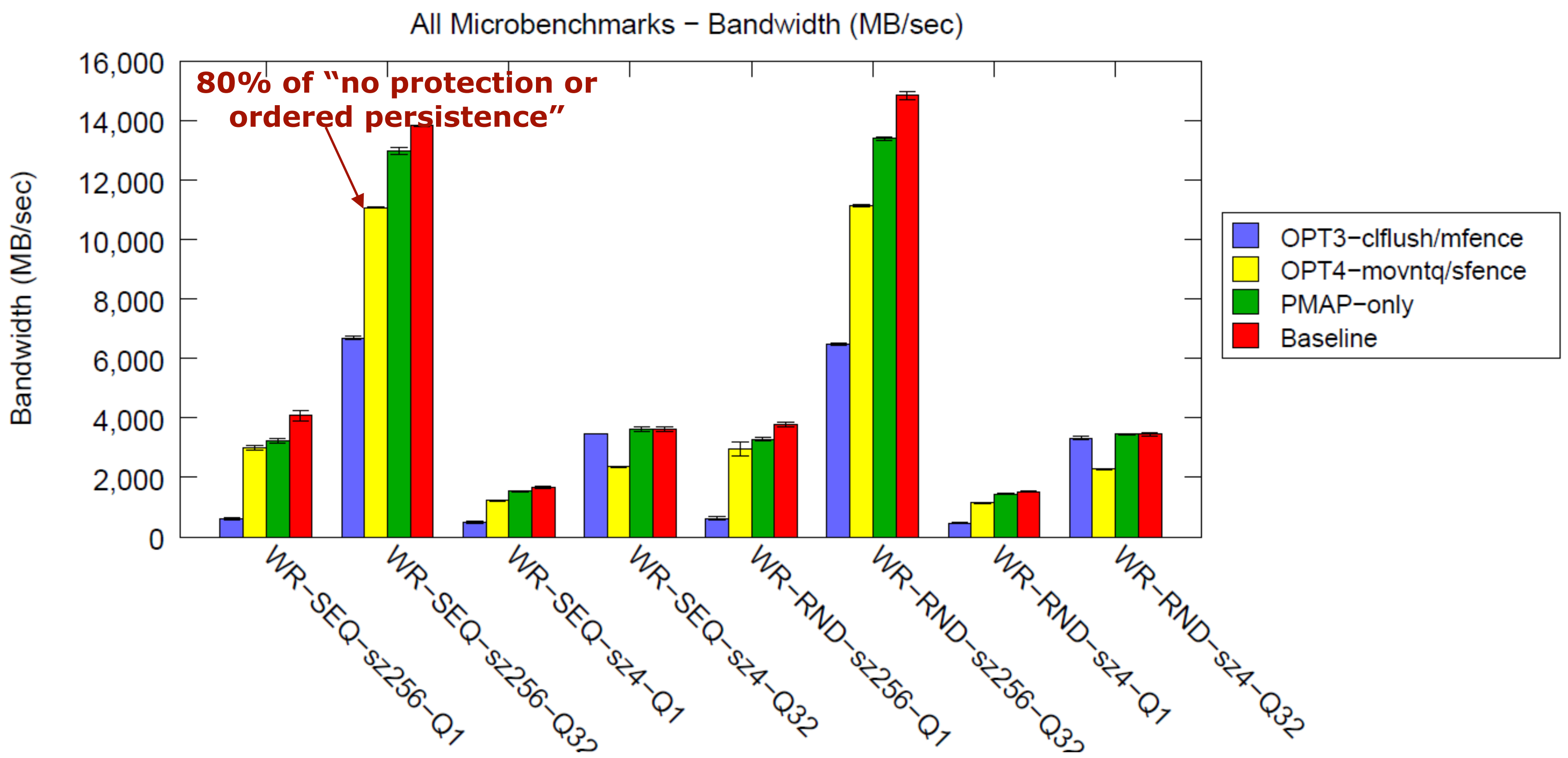
Applications and OS require support for <u>ordered</u> persistence

- Writes must complete in a specific order
  - The order of parallel writes being processed is random on the fly
  - Many applications rely on strict write ordering – e.g., database log
- The OS specifies the order (via write barrier), the device enforces it

Implications to PMBD design for persistence

- All prior writes must be completed (persistent) upon write barriers
- CPU cache effects must be addressed (like a disk cache)
  - Option 1 – Using *uncachable* or *write-through* – too slow
  - Option 2 – Flushing entire cache – ordinary stores, *wbinvd* in barriers
  - **Option 3** – Flushing cacheline after a write – ordinary stores, *clflush*/*mfence*
  - **Option 4** – Bypassing cache – NT store, movntq/*sfence* (***our recommendation***)

Performance of write schemes



- NT-store+sfence performs best in most cases – up to 80% of the performance upper bound (no protection/no ordered persistence)

Recalling our goals

- ✓ **Compatibility** – the block-based hybrid model
- ✓ **Protection** – private memory mapping for protection
- ✓ **Persistence** – non-temporal store + sfence + write barriers
- ✓ **Performance** – Low overhead for protection and persistence

Macro-benchmarks & system implications

Experimental setup

Xeon X5680 @ 3.3GHz (6 cores) x2

4GB main memory

PM (16GB DRAM)

OS – Fedora Core 13 (Linux 2.6.34)

File System – Ext4

Macrobenchmark workloads

name	Read Data (%)	Write Data (%)	Data Set Size (MBs)	Total Amount (MB)	Description
devel	61.1	38.9	2,033	3,470	FS sequence ops: untar, patch, tar, diff …
glimpseindex	94.5	5.5	12,504	6,019	Text indexing engine. Index 12GB linux source code files.
tar	53.1	46.9	11,949	11,493	Compressing 6GB linux kernel source files into one tar ball.
untar	47.8	52.2	11,970	11,413	Uncompressing a 6GB linux kernel tar ball
sfs-14g	92.6	7.4	11,210	146,674	SpecFS (14GB): 10,000 files, 500,000 transactions, 1,000 subdir.
tpch (all)	90.3	9.7	10,869	78,126	TPC-H Query (1-22): SF 4, PostgreSQL 9, 10GB data set
tpcc	36.2	63.9	11,298	98K-419K	TPC-C: PostgreSQL 9, 80 WH, 20 connections, 60 seconds
clamav	99.7	0.3	14,495	5,270	Virus scanning on 14GB files generated by SpecFS

Comparing to flash SSDs and hard drives

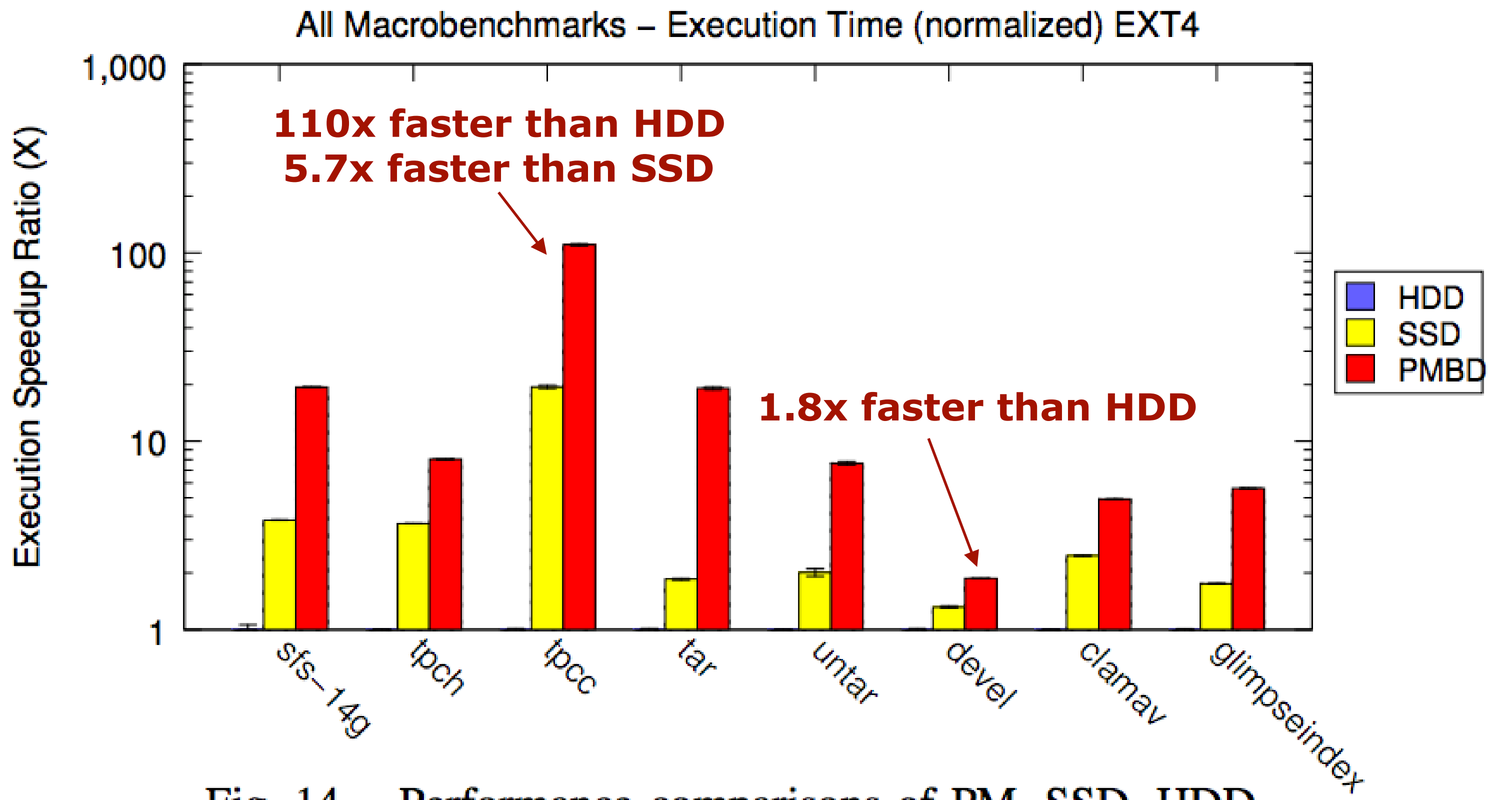
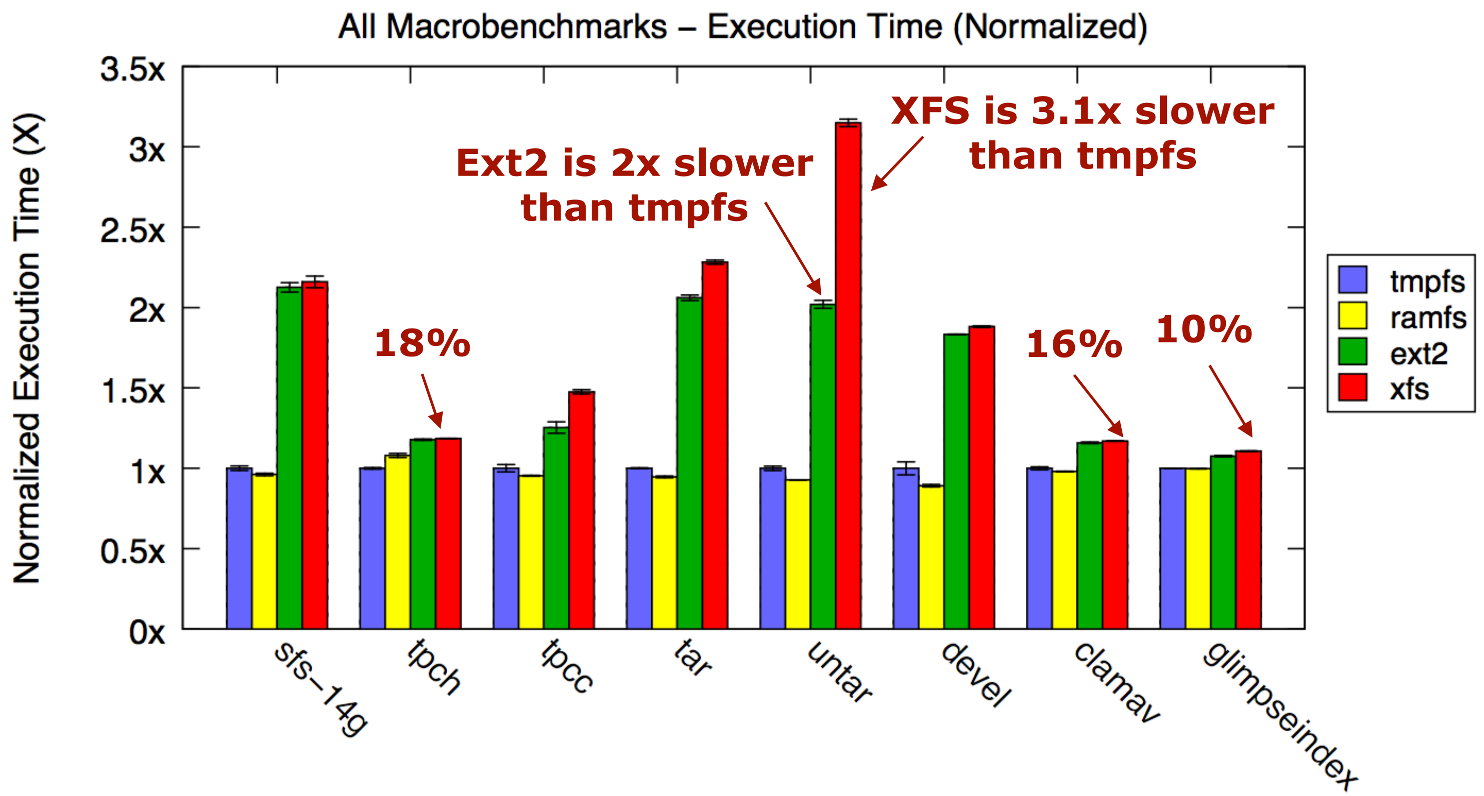


Fig. 14. Performance comparisons of PM, SSD, HDD

- PMBD outperforms flash SSDs and hard drives significantly
- Relatively performance speedup is workload dependent

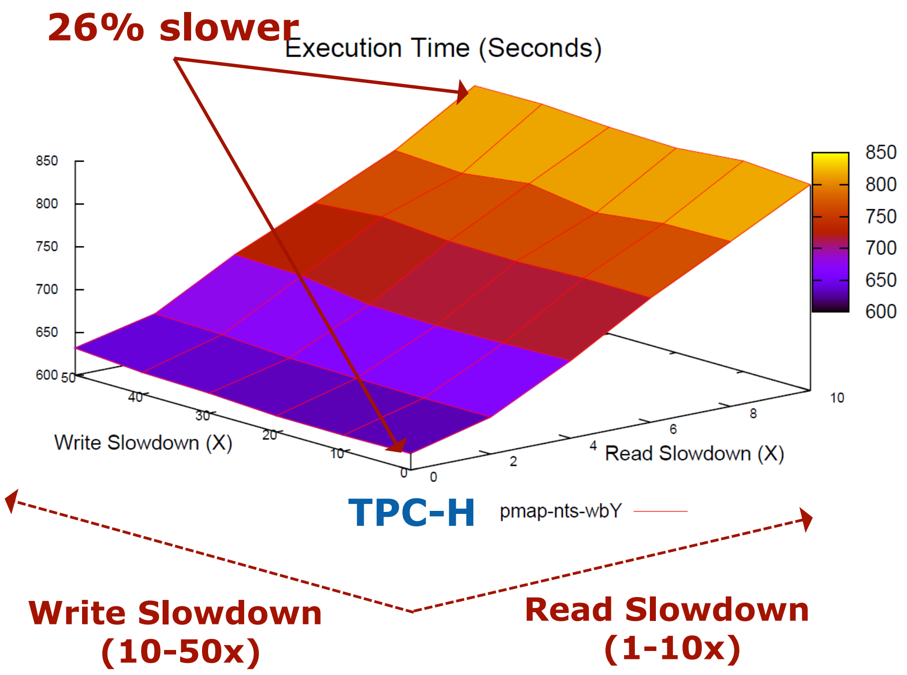
Comparing to memory-based file systems

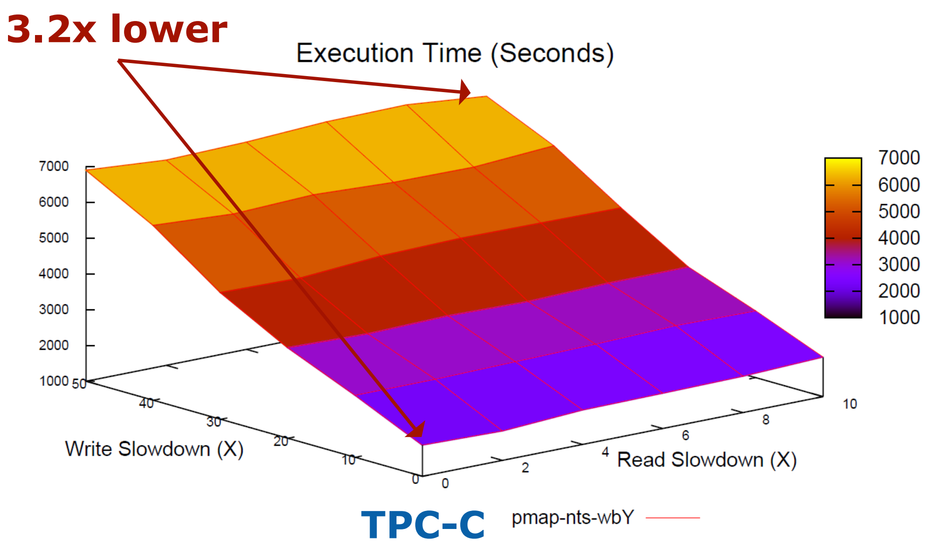


- tmpfs and ramfs outperforms legacy disk-based file systems on PMBD
    - Both provide no protection, no persistence, no journaling, and no extra memcpy
- Relative speedup is workload dependent and bounded (10%~3.1x)

A FS for PM could provide better performance, but actual benefits depend

Performance sensitivity to R/W asymmetry





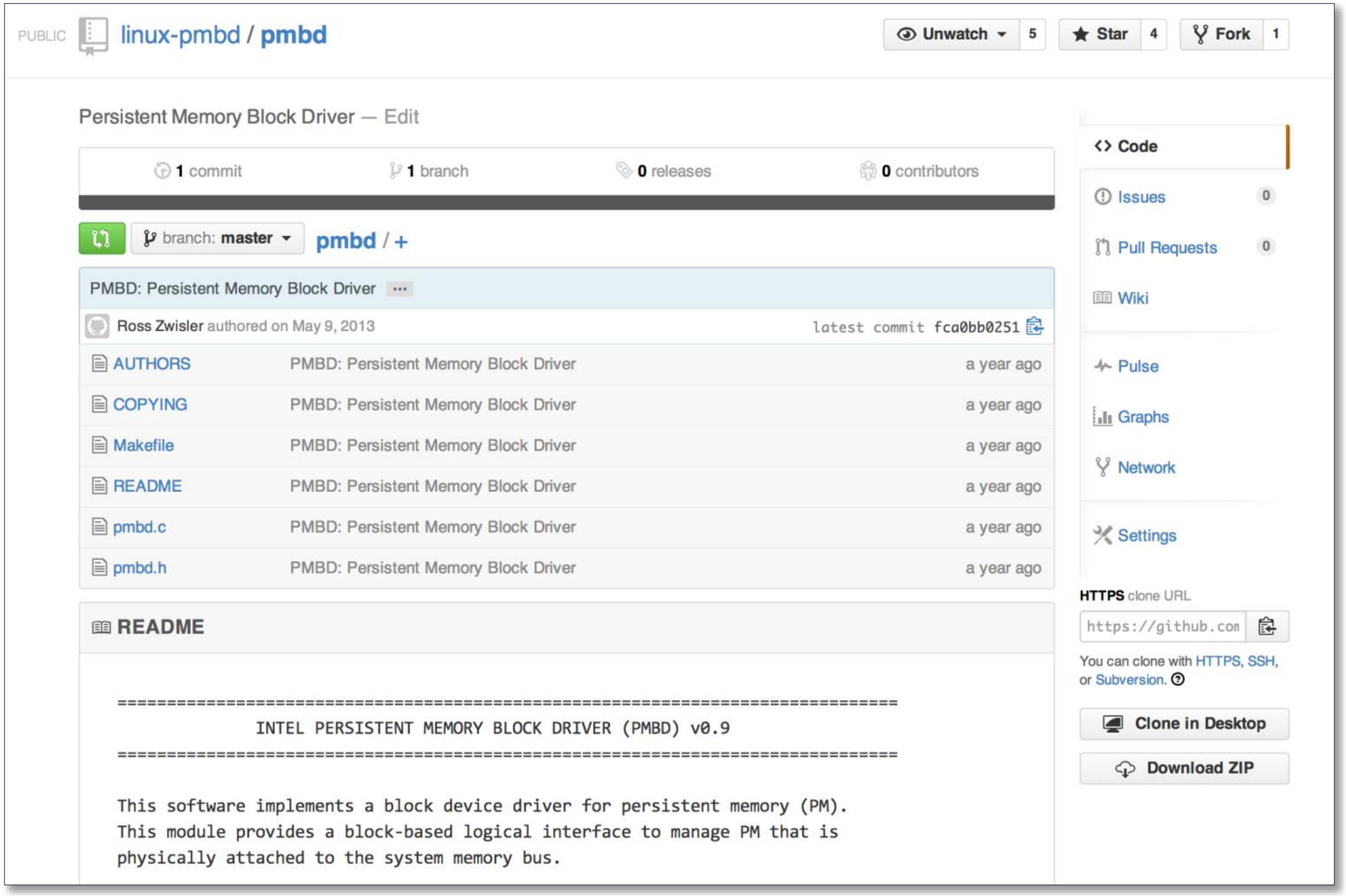
- PM speeds emulated by injecting delays proportional to DRAM speed
- App. performance is not proportional to read/write speed (TPC-H: 26%)
- Performance sensitivity is workload dependent (TPC-H: RD, TPC-C: WR)

Performance sensitivity to R/W asymmetry is highly workload dependent

Conclusions

- We propose **a hybrid model** for PM
  - Physically managed like memory, logically addressed like storage
- We have developed **a protected block device for PM (*PMBD*)**
  - Compatibility – a block device driver
  - Protection – private memory mapping
  - Persistence – non-temporal store + sfence + write barriers
  - Performance – performance close to raw memory performance
- Our **experimental studies** on PM show that
  - Protection and persistence can be achieved with relatively low overhead
  - FS and R/W asymmetry of PM affect application performance differently
  - PM performance can be well exploited with a hybrid solution with small overhead

PMBD: Open-source for public downloading



Persistent Memory Block Driver — Edit

1 commit | 1 branch | 0 releases | 0 contributors

branch: master pmbd / +

PMBD: Persistent Memory Block Driver ...

Ross Zwisler authored on May 9, 2013 latest commit fca0bb0251

AUTHORS	PMBD: Persistent Memory Block Driver	a year ago
COPYING	PMBD: Persistent Memory Block Driver	a year ago
Makefile	PMBD: Persistent Memory Block Driver	a year ago
README	PMBD: Persistent Memory Block Driver	a year ago
pmbd.c	PMBD: Persistent Memory Block Driver	a year ago
pmbd.h	PMBD: Persistent Memory Block Driver	a year ago

You can clone with HTTPS, SSH, or Subversion.

README

```
===============================================================================
              INTEL PERSISTENT MEMORY BLOCK DRIVER (PMBD) v0.9
===============================================================================

This software implements a block device driver for persistent memory (PM).
This module provides a block-based logical interface to manage PM that is
physically attached to the system memory bus.
```

https://github.com/linux-pmbd/pmbd

Thank you!

Contact:

fchen@csc.lsu.edu
michael.mesnier@intel.com
scott.hahn@intel.com